

Efficient Parallel and External Matching

Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, Nodari Sitchinava

Karlsruhe Institute of Technology, Karlsruhe, Germany

marcelbirn@gmx.de, {osipov,sanders,christian.schulz}@kit.edu, nodari@ira.uka.de

Abstract. We show that a simple algorithm for computing a matching on a graph runs in a logarithmic number of phases incurring work linear in the input size. The algorithm can be adapted to provide efficient algorithms in several models of computation, such as PRAM, External Memory, MapReduce and distributed memory models. Our CREW PRAM algorithm is the first $\mathcal{O}(\log^2 n)$ time, linear work algorithm. Our experimental results indicate the algorithm's high speed and efficiency combined with good solution quality.

1 Introduction

A matching M of a graph $G = (V, E)$ is a subset of edges such that no two elements of M have a common end point. Many applications require the computation of matchings with certain properties, like being maximal (no edge can be added to M without violating the matching property), having maximum cardinality, or having maximum total weight $\sum_{e \in M} w(e)$. Although these problems can be solved optimally in polynomial time, optimal algorithms are not fast enough for many applications involving large graphs where we need near linear time algorithms. For example, the most efficient algorithms for graph partitioning rely on repeatedly contracting maximal matchings, often trying to maximize some edge rating function w . Refer to [11] for details and examples. For very large graphs, even linear time is not enough – we need a parallel algorithm with near linear work or an algorithm working in the external memory model [23].

Here we consider the following simple *local max* algorithm [10]: Call an edge locally maximal, if its weight is larger than the weight of any of its incident edges; for unweighted problems, assign unit weights to the edges. When comparing edges of equal weight, use tie breaking based on random perturbations of the edge weights. The algorithm starts with an empty matching M . It repeatedly adds locally maximal edges to M and removes their incident edges until no edges are left in the graph. The result is obviously a maximal matching (every edge is either in M or it has been removed because it is incident to a matched edge). The algorithm falls into a family of weighted matching algorithms for which Preis [21] shows that they compute a $1/2$ -approximation of the maximum weight matching problem. Hoepman [10] derives the local max algorithm as a distributed adaptation of Preis' idea. Based on this, Manne and Bisseling [16] devise sequential and parallel implementations. They prove that the algorithm needs only a logarithmic number of iterations to compute maximal matchings

by noticing that a maximal matching problem can be translated into a maximal independent set problem on the *line graph* which can be solved by Luby’s algorithm [15]. However, this does not yield an algorithm with linear work since it is not proven that the edge set indeed shrinks geometrically.¹ Manne and Bisseling also give a sequential algorithm running in time $\mathcal{O}(m \log \Delta)$ where Δ is the maximum degree. On a NUMA shared memory machine with 32 processors (SGI Origin 3800) they get relative speedup < 6 for a complete graph and relative speedup ≈ 10 for a more sparse graph partitioned with Metis. Since this graph still has average degree ≈ 200 and since the speedups are not impressive this is a somewhat inconclusive result when one is interested in partitioning large sparse graphs on a larger number of processors.

Parallel matching algorithms have been widely studied. There is even a book on the subject [14] but most theoretical results concentrate on work-inefficient algorithms. The only linear work parallel algorithm that we are aware of is a randomized CRCW PRAM algorithm by Israeli and Itai [12] which runs in $\mathcal{O}(\log n)$ time and incurs linear work. Their algorithm, which we call IIM, provably removes a constant fraction of edges in each iteration.

Fagginger Auer and Bisseling [6] study an algorithm similar to [12] which we call red-blue matching (RBM) here. They implement RBM on shared memory machines and GPUs. They prove good shrinking behavior for random graphs, however, provide no analysis for arbitrary graphs.

Our contributions. We give a simple approach to implementing the local max algorithm that is easy to adapt to many models of computation. We show that for computing maximal matchings, the algorithm needs only linear work on a sequential machine and in several models of parallel computation (Section 2). Moreover it has low I/O complexity on several models of memory hierarchies.

Our CRCW PRAM local max algorithm matches the optimal asymptotic bounds of IIM. However, our algorithm is simpler (resulting in better constant factors), removes higher fraction of edges in each iteration (IIM’s proof shows less than 5% per iteration, while we show at least 50%) and our analysis is a lot simpler. We also provide the first CREW PRAM algorithm which runs in $\mathcal{O}(\log^2 n)$ time and linear work.²

In Section 3 we explain how to implement local max on practical massively parallel machines such as MPI clusters and GPUs. Our experiments indicate that the algorithm yields surprisingly good quality for the weighted matching problem and runs very efficiently on sequential machines, clusters with reasonably partitioned input graphs, and on GPUs. Compared to RBM, the local max implementations remove more edges in each iteration and provide better quality results for the weighted case. Some of the results presented here are from the diploma thesis of Marcel Birn [2].

¹ Manne and Bisseling show such a shrinking property under an assumption that unfortunately does not hold for all graphs.

² While a generic simulation of IIM on the CREW PRAM model will result in a $\mathcal{O}(\log^2 n)$ time algorithm, the simulation incurs $\mathcal{O}(n \log n)$ work due to sorting.

2 Parallel Local Max

Our central observation is:

Lemma 1. *Each iteration of the local max algorithm for the unit weight case removes at least half of the edges in expectation.*

Proof. Consider the graph remaining in the currently considered iteration where $d(v)$ denotes the degree of a node and m the remaining number of edges. Consider the end point at node v of an edge $\{u, v\}$ as *marked* if and only if some edge incident to v becomes matched. Note that an edge is removed if and only if at least one of its end points becomes marked. Now consider a particular edge $e = \{u, v\}$. Since any of the $d(u) + d(v) - 1$ edges incident to u and v is equally likely to be locally maximal, e becomes matched with probability $1/(d(u) + d(v) - 1)$.³ If e is matched, this event is responsible for setting $d(u) + d(v)$ marks, i.e., the expected number of marks caused by an edge is $(d(u) + d(v))/(d(u) + d(v) - 1) \geq 1$. By linearity of expectation, the total expected number of marks is at least m . Since no edge can have more than two marks, at least $m/2$ edges have at least one mark and are thus deleted.⁴ ■

Assume now that each iteration can be implemented to run with work linear in the number of surviving edges (independent of the number of nodes). Working naively with the expectations, this gives us a logarithmic number of rounds and a geometric sum leading to linear total work for computing a maximal matching. This can be made rigorous by distinguishing *good* rounds with at least $m/4$ matched edges and bad rounds with less matched edges. By Markov's inequality, we have a good round with constant probability. This is already sufficient to show expected linear work and a logarithmic number of expected rounds. We skip the details since this is a standard proof technique and since the resulting constant factors are unrealistically conservative. An analogous calculation for median selection can be found in [18, Theorem 5.8]. One could attempt to show a shrinking factor close to $1/2$ rigorously by showing that large deviations (in the wrong direction) from the expectation are unlikely (e.g., using Martingale tail bounds). However this would still be a factor two away from the more heuristic argument in Footnote 4 and thus we stick to the simple argument.

There are many ways to implement an iteration which of course depend on the considered model of computation.

Sequential Model. For each node v maintain a candidate edge $C[v]$, originally initialized to a dummy edge with zero weight. In an iteration go through all remaining edges $e = \{u, v\}$ three times. In the first pass, if $w(e) > w(C[u])$ set $C[u] := e$ (add random perturbation to $w(e)$ in case of a tie). If $w(e) > w(C[v])$

³ For this to be true, the random noise added for tie breaking needs to be renewed in every iteration. However, in our experiments this had no noticeable effect.

⁴ This is a conservative estimate. Indeed, if we make the (over)simplified assumption that m marks are assigned randomly and independently to $2m$ end points, then only one fourth of the edges survives in expectation. Interestingly, this is the amount of reduction we observe in practice – even for the weighted case.

set $C[v] := e$. In the second pass, if $C[u] = C[v] = e$ put e into the matching M . In the third pass, if u or v is matched, remove e from the graph. Otherwise, reset the candidate edge of u and v to the dummy edge. Note that except for the initialization of C which happens only once before the first iteration, this algorithm has no component depending on the number of nodes and thus leads to linear running time in total if Lemma 1 is applied.

CRCW PRAM Model. In the most powerful variant of the *Combining CRCW PRAM* that allows concurrent writes with a maximum reduction for resolving write conflicts, the sequential algorithm can be parallelized directly running in constant time per iteration using m processors.

MapReduce Model. The CRCW PRAM result together with the simulation result of Goodrich et al. [9] immediately implies that each iteration of local max can be implemented in $\mathcal{O}(\log_M n)$ rounds and $\mathcal{O}(m \log_M n)$ communication complexity in the MapReduce model, where M is the size of memory of each compute node. Since typical compute nodes in MapReduce have at least $\Omega(m^\epsilon)$ memory [13], for some constant $\epsilon > 0$, each iteration of local max can be performed in MapReduce in constant rounds and linear communication complexity.

External Memory Models. Using the PRAM emulation techniques for algorithms with geometrically decreasing input size from [3, Theorem 3.2] the above algorithm can be implemented in the external memory [1] and cache-oblivious [7] models in $\mathcal{O}(\text{sort}(m))$ I/O complexity, which seems to be optimal.

2.1 $\mathcal{O}(\log^2 n)$ work-optimal CREW solution

In this section, we present a $\mathcal{O}(\log^2 n)$ CREW PRAM algorithm, which incurs only $\mathcal{O}(n + m)$ work.

Converting one representation of a graph into another can require as many as $\Omega(m \log m)$ operations (e.g. the conversion from an unordered list of edges into adjacency list representation requires sorting the edges). To perform matching in $\mathcal{O}(n + m)$ work, we define a graph representation suitable for our algorithm.

Consider an array \mathbf{V} of n elements, where each entry $\mathbf{V}[i] = \sum_{j < i} \deg(v_j)$ ($\deg(v_j)$ denotes the degree of node v_j). An adjacency array representation of a graph is an array \mathbf{A} , where entries $\mathbf{A}[\mathbf{V}[i]]$ through $\mathbf{A}[\mathbf{V}[i + 1] - 1]$ are associated with vertex $v_i \in G$ and store the edges incident on v_i .

We consider the following slightly altered adjacency array representation: the edges are stored in a separate array \mathbf{E} and the entries $\mathbf{A}[\mathbf{V}[i]]$ through $\mathbf{A}[\mathbf{V}[i + 1] - 1]$ store the *pointers* to the corresponding edges in \mathbf{E} (see Figure 1). Thus, we can view each entry of \mathbf{A} as a tuple (v, e_k) , where v is a node in G and e_k is a pointer to a record $\mathbf{E}[k]$ with information about the edge incident on v , such as the two vertices of the edge, edge weight, or any other auxiliary information.

Note that any edge $\mathbf{E}[k] = \{v_i, v_j\}$ contains two corresponding entries in \mathbf{A} pointing to it: (v_i, e_k) and (v_j, e_k) . During our algorithm, a processor responsible for (v_i, e_k) might need to find and update entry (v_j, e_k) (and vice versa). The following lemma describes how to compute for each entry (v_i, e_k) the address of the corresponding entry (v_j, e_k) in \mathbf{A} .

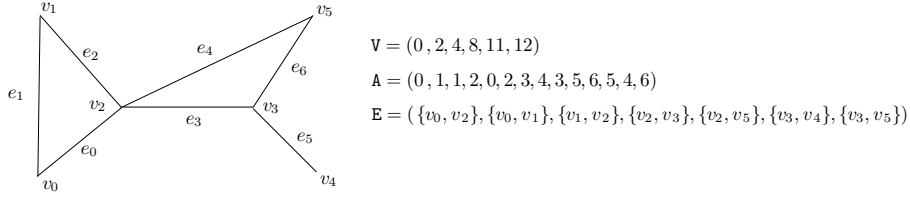


Fig. 1. Adjacency representation of a graph: Array E is a collection of edges. Entries of $A[V[i]]$ through $A[V[i + 1] - 1]$ point to edges in E incident on vertex v_i .

Lemma 2. For every edge $e_k = \{v_i, v_j\} \in E$ entries (v_i, e_k) and (v_j, e_k) of A can compute each other's index in A in $\mathcal{O}(1)$ time and $\mathcal{O}(|A|)$ work in the CREW PRAM model.

Proof. For every $E[k] = \{v_i, v_j\}$ we show how (v_j, e_k) can compute the address of the corresponding entry (v_i, e_k) in V for $i < j$. The addresses for the other half of the entries are computed symmetrically.

The algorithm proceeds in two phases. In the first phase, each entry (v_i, e_k) , writes the address of (v_i, e_k) in $E[k] = \{v_i, v_j\}$, iff $i < j$. In the second phase, each entry (v_j, e_k) reads the address of (v_i, e_k) from $E[k] = \{v_i, v_j\}$ iff $j > i$.

If we assign a separate processor to each entry of A , each processor performs only $\mathcal{O}(1)$ steps. Moreover, there are no concurrent writes because, at each step only one of the two vertices of the edge e_k writes to $E[k]$. Note, we need a concurrent read to $E[k] = \{v_i, v_j\}$ to determine the relative order of i and j for v_i and v_j . ■

Lemma 3. Using our graph representation, each node v in the graph can apply an associative operator \oplus to all edges incident on v in $\mathcal{O}(\log |A|)$ time and $\mathcal{O}(|A|)$ work on the CREW PRAM model.

Proof. First, we read for each entry $(v, e_k) \in A$ the value from $E[k]$ on which to apply the operator. Next, we run segmented prefix sums with \oplus operator on these values, where segments are the portions of A representing the neighbors of a single node. Finally, each entry of $(v, e_k) \in A$ applies its result of segmented prefix sums to the edge $E[k]$, while using the technique of Lemma 2 to avoid write conflicts. Each step of the algorithm can be implemented in $\mathcal{O}(\log |A|)$ time using $\mathcal{O}(|A|)$ work. ■

Now we are ready to describe the solution to the matching problem. We perform the following in each phase of the local max algorithm.

1. Each edge $e_k \in E$ picks a random weight w_k .
2. Using Lemma 3, each vertex v identifies the heaviest edge e_k incident on v by applying the associative operator MAX to the edge weights picked in the previous step.

3. Using Lemma 2, each entry (v_i, e_k) checks if $E[k] = \{v_i, v_j\}$ is also the heaviest incident edge on v_j . If so and $i < j$, v_i adds e_k to the matching and sets the deletion flag $f = 1$ on $E[k]$.
4. Using Lemma 3, each entry (v_i, e_k) spreads the deletion flag over all edges incident on v_i by applying MAX associative operator on the deletion flags of incident edges on v_i . Thus, if at least one edge incident on v_i was added to the matching, all edges incident on v_i will be marked for deletion.
5. Now we must prepare the graph representation for the next phase by removing all entries of E and A marked for deletion, compacting E and A and updating the pointers of A to point to the compacted entries of E . To perform the compaction, we compute for each entry $E[k]$, how many entries $E[i]$ and $A[i]$, $i \leq k$ must be deleted. This can be accomplished using parallel prefix sums on the deletion flags of each entry in E and A . Let the result of prefix sums for edge $E[k]$ be d_k and for entry $A[i]$ be r_i . Then $k - d_k$ is the new address of the entry $E[k]$ and $i - r_i$ is the new address of $A[i]$ once all edges marked for deletion are removed.
6. Each entry $E[k]$ that is not marked for deletion copies itself to $E[k - d_k]$. The corresponding entry $(v, e_k) \in A$ updates itself to point to the new entry $E[k - d_k]$, i.e., (v, e_k) becomes (v, e_{k-d_k}) , and copies itself to $A[i - r_i]$.

The algorithm defines a single phase of the local max algorithm. Each step of the phase takes at most $\mathcal{O}(\log(m + n)) = \mathcal{O}(\log n)$ time and $\mathcal{O}(n + m)$ work in the CREW PRAM model. Over $\mathcal{O}(\log m)$ phases, each with geometrically decreasing number of edges, the local max takes $\mathcal{O}(\log^2 n)$ time and $\mathcal{O}(n + m)$ work in the CREW PRAM model.

3 Implementations and Experiments

We now report experiments focusing on computing approximate maximum weight matchings. We consider the following families of inputs, where the first two classes allow comparison with the experiments from [17].

Delaunay Instances are created by randomly choosing $n = 2^x$ points in the unit square and computing their Delaunay triangulation. Edge weights are Euclidean distances.

Random graphs with $n := 2^x$ nodes, αn edges for $\alpha = \{4, 16, 64\}$, and random edge weight chosen uniformly from $[0, 1]$.

Random geometric graphs with 2^x nodes (rggx). Each vertex is a random point in the unit square and edges connect vertices whose Euclidean distance is below $0.55 \ln n/n$. This threshold was chosen in order to ensure that the graph is almost connected.

Florida Sparse Matrix. Following [6] we use 126 symmetric non-0/1 matrices from [4] using absolute values of their entries as edge weights, see Appendix for the full list. The number of edges of the resulting graphs $m \in (0.5 \dots 16) \times 10^6$. See Appendix B for a detailed list.

Graph Contraction. We use the graphs considered by KaFFPa for partitioning graphs from the 10'th DIMACS Implementation Challenge [22].

We compare implementations of local max, the red-blue algorithm from [6] (RBM) (their implementation), heavy edge matching (HEM) [8], greedy, and the global path algorithm (GPA) [17]. HEM iterates through the nodes (optionally in random order) and matches the heaviest incident edge that is nonadjacent to a previously matched edge. The greedy algorithm sorts the edges by decreasing weights, scans them and inserts edges connecting unmatched nodes into the matching. GPA refines greedy. It greedily inserts edges into a graph G_2 with maximum degree two and no odd cycles. Using dynamic programming on the resulting paths and even cycles, a maximum weight matching of G_2 is computed.

Sequential and shared-memory parallel experiments were performed on an Intel i7 920 2.67 GHz quad-core machine with 6 GB of memory. We used a commodity NVidia Fermi GTX 480 featuring 15 multiprocessors, each containing 32 scalar processors, for a total of 480 CUDA cores on chip. The GPU RAM is 1.5 GB. We compiled all implementations using CUDA 4.2 and Microsoft Visual Studio 2010 on 64-bit Windows 7 Enterprise with maximum optimization level.

3.1 Sequential Speed and Quality

We compare solution quality of the algorithms relative to GPA. Via the experiments in [17] this also allows some comparison with optimal solutions which are only a few percent better there. Figure 2 shows the quality for Delaunay graphs (where GPA is about 5 % from optimal [17]). We see that local max achieves almost the same quality as greedy which is only about 2 % worse than GPA. HEM, possibly the fastest nontrivial sequential algorithm is about 13 % away while RBM is 14 % worse than GPA, i.e., HEM and RBM almost double the gap to optimality of local max. Looking at the running times, we see that

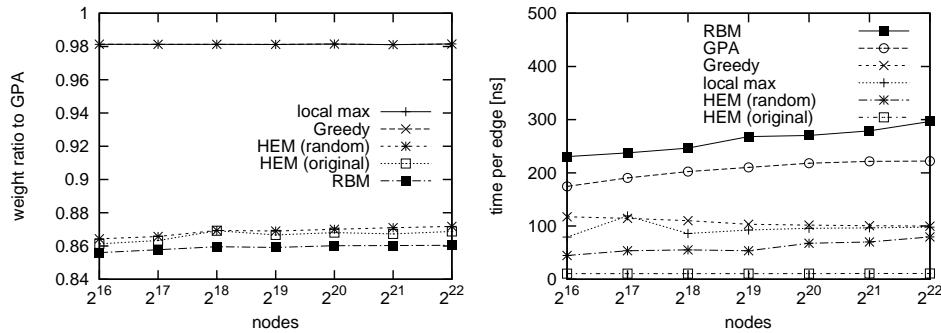


Fig. 2. Ratio of the weights computed by GPA and other algorithms for Delaunay instances and running times.

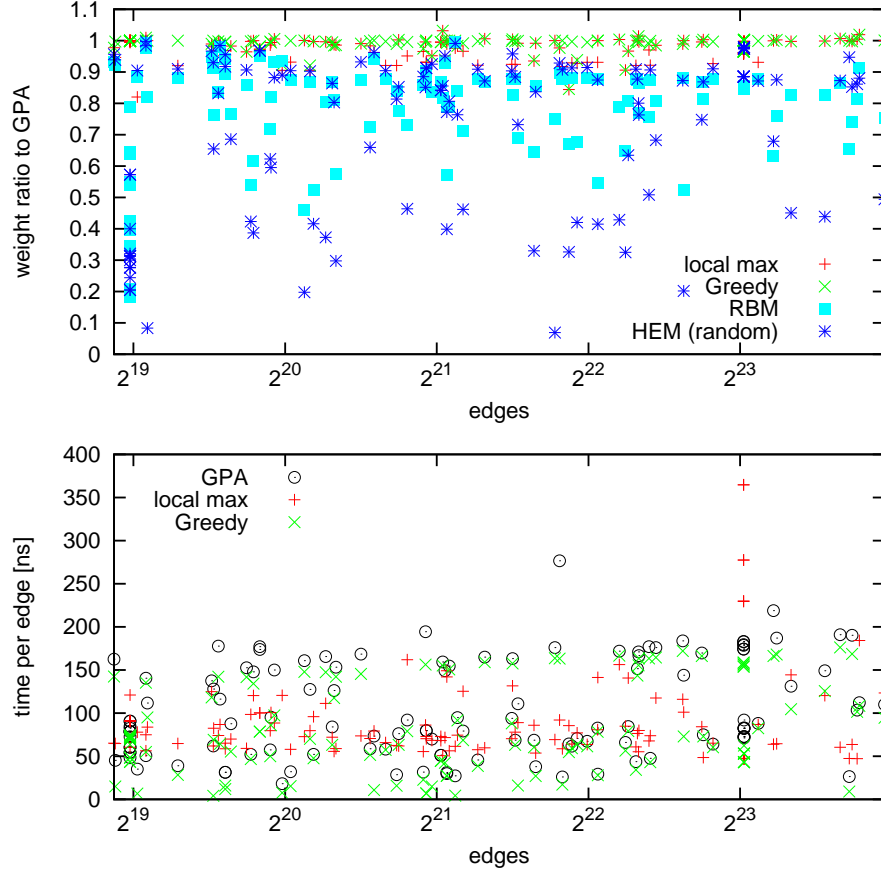


Fig. 3. Ratio of the weights computed by GPA and other sequential algorithms for sparse matrix instances and running time.

HEM is the fastest (with a surprisingly large cost for actually randomizing node orders) followed by local max, greedy, GPA, and RBM. From this it looks like HEM, local max, and GPA are the winners in the sense that none of them is dominated by another algorithm with respect to both quality and running time. Greedy has similar quality as local max but takes somewhat longer and is not so easy to parallelize. RBM as a sequential algorithm is dominated by all other algorithms. Perhaps the most surprising thing is that RBM is fairly slow. This has to be taken into account when evaluating reported speedups. We suspect that a more efficient implementation is possible but do not expect that this changes the overall conclusion. In Appendix A we report similar results for the rgg instances (Figure 6) and random graphs (Figures 7, 8, 9).

Looking at the wide range of instances in the Florida Sparse Matrix collection leads to similar but more complicated conclusions. Figure 3 shows the solution qualities for greedy, local max, RBM and HEM relative to GPA. RBM and even more so HEM shows erratic behavior with respect to solution quality. Greedy and local max are again very close to GPA and even closer to each other although there is a sizable minority of instances where greedy is somewhat better than local max. Looking at the corresponding running times one gets a surprisingly diverse picture. HEM which is again fastest and RBM which is again dominated by local max are not shown. There are instances where local max is considerably faster than greedy and vice versa. A possible explanation is that greedy becomes quite fast when there is only a small number of different edge weights since then sorting is a quite easy problem.

Experiments on the graph contraction instances in [2] show local max about 1 % away from GPA. For these instances the average fraction of remaining edges after an iteration is well below 25 %. Notable exceptions are the graphs *add20* and *memplus* which both represent VLSI circuits. Nevertheless, none of the instances considered required more than 10 iterations.

3.2 Distributed Memory Implementation

Our distributed memory parallelization (using MPI) on p processing elements (PEs or MPI processes) assigns nodes to PEs and stores all edges incident to a node locally. This can be done in a load balanced way if no node has degree exceeding m/p . The second pass of the basic algorithm from Section 2 has to exchange information on candidate edges that cross a PE boundary. In the worst case, this can involve all edges handled by a PE, i.e., we can expect better performance if we manage to keep most edges locally. In our experiments, one

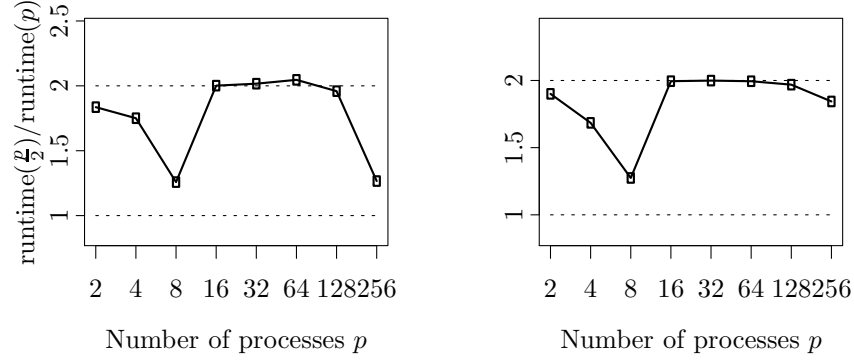


Fig. 4. Scaling results of the parallel local max algorithm on random geometric graphs with random edge weights. Left: rgg23 (≈ 63 million edges). Right: rgg24 (≈ 132 million edges).

PE owns nodes whose numbers are a consecutive range of the input numbers. Thus, depending on how much locality the input numbering contains we have a highly local or a highly non-local situation. We have not considered more sophisticated ways of node assignment so far since our motivating application is graph partitioning/clustering where almost by definition we initially do *not* know which nodes form clusters – this is the intended *output*. Since Lemma 1 also applies to the subgraph relevant for a particular PE, we can expect that the graph shrinks fairly uniformly over the entire network.

We performed experiments on two different clusters at the KIT computing center both using compute-nodes with two quad-core processors each. Refer to [2] for details. We ran experiments with up to 128 compute-nodes corresponding to 1024 cores with one MPI process per core.

Figure 4 illustrates how our distributed local max implementation scales for the random geometric graphs *rgg23* and *rgg24* (using random edge weights) which have fairly good locality. We plot the decrease in running time for successive doubling of p , i.e., a value of two stands for perfect relative speedup for this step and a value below one means that parallelization no longer helps. We see values slightly below two for the steps $1 \rightarrow 2$ and $2 \rightarrow 4$ which is typical behavior of multicore algorithms when cores compete for resources like memory bandwidth. For $p = 8$ we start to use two compute-nodes (with 4 active cores each) and consequently we see the largest dip in efficiency. Beyond that, we have almost perfect scaling until the problem instance becomes too small. We have similar behavior for other graphs with good locality. For graphs with poor locality, efficiency is not very good. However the ratios stay above one for a very long time, i.e., it pays to use parallelism when it is available anyway. This is the situation we have when partitioning large graphs for use on massively parallel machines. Considering that the matching step in graph partitioning is often the least work intensive one in multi-level graph partitioning algorithms we conclude that local max might be a way to remove a sequential bottleneck from massively parallel graph partitioning. Refer to [2] for additional data.

3.3 GPU Implementation

Our GPU algorithm is a fairly direct implementation of the CRCW algorithm. We reduce the algorithm to the basic primitives such as segmented prefix sum, prefix sum and random gather/scatter from/to GPU memory. As a basis for our implementation we use back40computing library by Merrill [19].

Figure 5 compares the running time of our implementation with GPA, sequential local max, the RBM algorithm parallelized for 4 cores, and its GPU parallelization from [6]. While the CPU implementation has troubles recovering from its sequential inefficiency and is only slightly faster than even sequential local max, the GPU implementation is impressively fast in particular for small graphs. For large graphs, the GPU implementation of local max is faster. Since local max has better solution quality, we consider this a good result. Our GPU code is up to 35 times faster than sequential local max. We may also be able

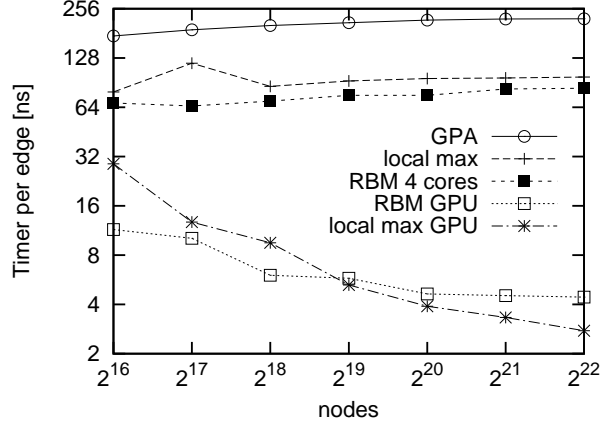


Fig. 5. Running time of sequential and GPU algorithms for Delaunay instances.

to learn from the implementation techniques of RBM GPU for small inputs in future work.

For random geometric graphs and random graphs, we get similar behavior (see Figure 10 and 11 in Appendix A). The results for rgg are slightly worse for GPU local max – speedup is up to 24 over sequential local max and a speed advantage over GPU RBM only for the very largest inputs. As for random graphs, the denser the graph is the larger is our speedup over the sequential and GPU RBM implementations. Thus, for $\alpha = 64$ our implementation is faster than GPU RBM for $n = 2^{15}$ already. While for $n = 2^{18}$ it is 65% faster than GPU RBM and 30 times faster than the sequential local max.

4 Conclusions

The local max algorithm is a good choice for parallel or external computation of maximal and approximate maximum weight matchings. On the theoretical side it is provably efficient for computing maximal matchings and guarantees a $1/2$ -approximation. On the practical side it yields better quality at faster speed than several competitors including the greedy algorithm and RBM. Somewhat surprisingly it is even attractive as a sequential algorithm, outperforming HEM with respect to solution quality and other algorithms with respect to speed.

Many interesting question remain. Can we omit re-randomization of edge weights when computing maximal matchings? Is there a linear work parallel algorithm with polylogarithmic execution time that computes $1/2$ -approximations (or any other constant factor approximation). Can we even do $2/3$ -approximations with linear work in parallel [5,20]?

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. Marcel Birn. Engineering fast parallel matching algorithms. Diploma Thesis, Karlsruhe Institute of Technology, 2012.
3. Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren E. Vengroff, and Jeffrey S. Vitter. External-memory graph algorithms. In *SODA*, pages 139–149, 1995.
4. T. Davis. The University of Florida Sparse Matrix Collection, <http://www.cise.ufl.edu/research/sparse/matrices>, 2008.
5. D. Drake and S. Hougardy. Improved linear time approximation algorithms for weighted matchings. In *APPROX, LNCS 2764*, pages 14–23, 2003.
6. B. O. Fagginger Auer and R. H. Bisseling. A GPU algorithm for greedy graph matching. In *Facing the Multicore - Challenge II*, pages 108–119, 2012.
7. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *FOCS*, pages 285–298, 1999.
8. V. Kumar G. Karypis. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
9. M. T. Goodrich, N. Sitchinava, and Q. Zhang. Sorting, searching and simulation in the mapreduce framework. In *ISAAC*, pages 374–383, 2011.
10. Jaap-Henk Hoepman. Simple distributed weighted matchings. *CoRR*, cs.DC/0410047, 2004.
11. M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *IPDPS*, pages 1–12, 2010.
12. Amos Israeli and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Information Processing Letters*, 22(2):77–80, 1986.
13. Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *SODA*, pages 938–948, 2010.
14. M. Karpinski and W. Rytter. *Fast parallel algorithms for graph matching problems*, volume 98. Clarendon Press, 1998.
15. M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.
16. F. Manne and R. H. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *PPAM*, volume 4967, pages 708–717, 2007.
17. J. Maue and P. Sanders. Engineering algorithms for approximate weighted matching. In *6th Workshop on Exp. Algorithms (WEA)*, 2007.
18. K. Mehlhorn and P. Sanders. *Algorithms and Data Structures — The Basic Toolbox*. Springer, 2008.
19. Duane Merrill. back40computing: fast and efficient software primitives for GPU computing. <http://code.google.com/p/back40computing/>.
20. S. Pettie and P. Sanders. A simpler linear time $2/3 - \epsilon$ approximation for maximum weight matching. Technical Report MPI-I-2004-1-002, MPII, 2004.
21. R. Preis. Linear time $1/2$ -approximation algorithm for maximum weighted matching in general graphs. In *STACS*, pages 259–269, 1999.
22. P. Sanders and C. Schulz. High quality graph partitioning. In *10th DIMACS Implementation Challenge – Graph Partitioning and Graph Clustering*. 2013. To appear.
23. J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two level memories. *Algorithmica*, 12(2/3):110–147, 1994.

A More Experimental results

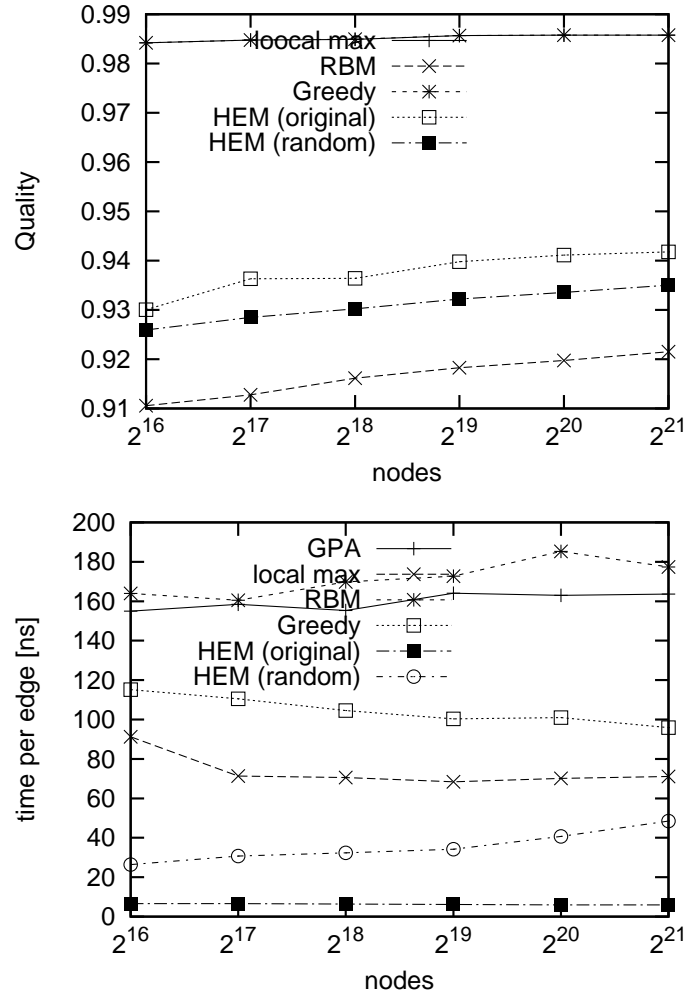


Fig. 6. Ratio of the weights computed by GPA and other algorithms for random geometric graphs and running time.

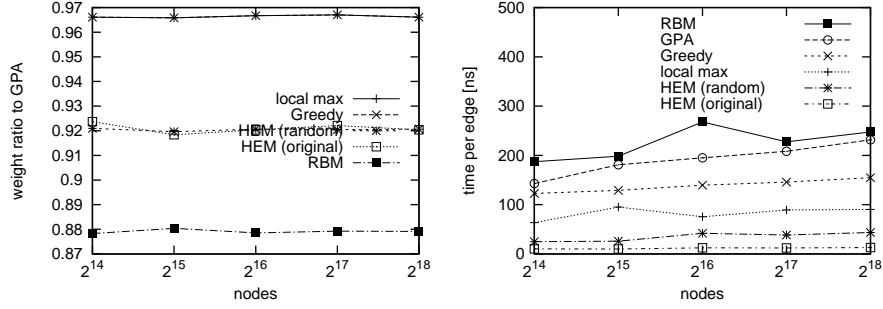


Fig. 7. Ratio of the weights computed by GPA and other sequential algorithms (left) and their timing (right) for random graphs with $\alpha = 4$.

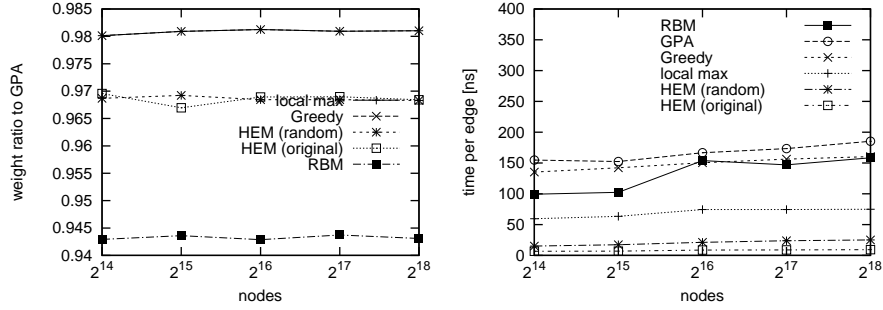


Fig. 8. Ratio of the weights computed by GPA and other sequential algorithms (left) and their timing (right) for random graphs with $\alpha = 16$.

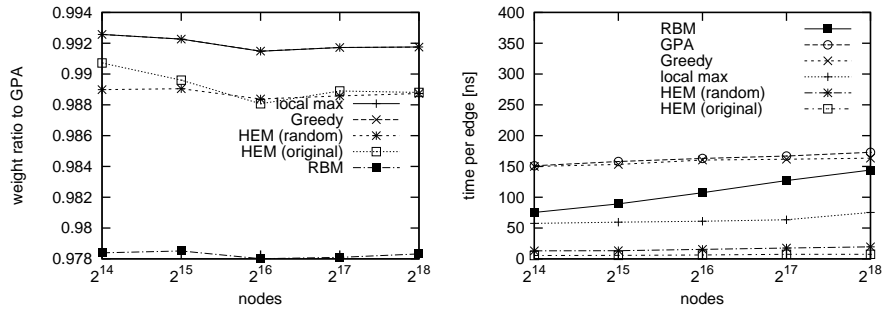


Fig. 9. Ratio of the weights computed by GPA and other sequential algorithms (left) and their timing (right) for random graphs with $\alpha = 64$.

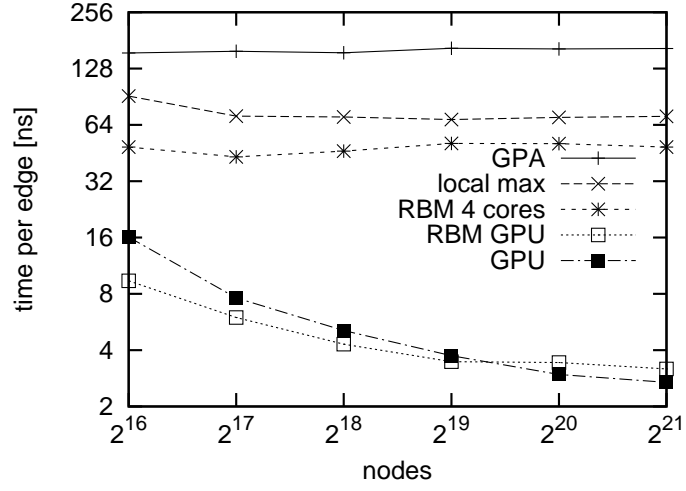


Fig. 10. Running time of sequential and GPU algorithms for random geometric graphs instances.

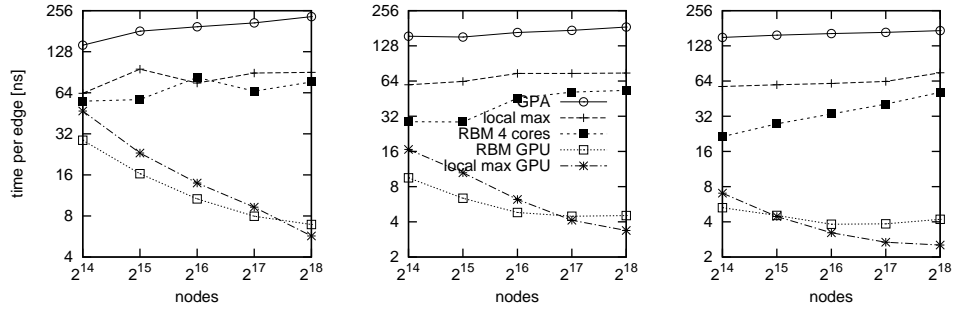


Fig. 11. Running time of sequential and GPU algorithms for random graphs with $\alpha = 4, 16, 64$ (from left to right).

B List of Instances

file	n	m	file	n	m
2cubes_sphere.mtx.graph	101492	772886	darcy003.mtx.graph	389874	933557
af_0_k101.mtx.graph	503625	8523525	dawson5.mtx.graph	51537	479620
af_1_k101.mtx.graph	503625	8523525	denormal.mtx.graph	89400	533412
af_2_k101.mtx.graph	503625	8523525	dielFilterV2clx.mtx.graph	607232	12351020
af_3_k101.mtx.graph	503625	8523525	dielFilterV3clx.mtx.graph	420408	16232900
af_4_k101.mtx.graph	503625	8523525	Dubcova2.mtx.graph	65025	482600
af_5_k101.mtx.graph	503625	8523525	Dubcova3.mtx.graph	146689	1744980
af_shell1.mtx.graph	504855	8542010	d_pretok.mtx.graph	182730	756256
af_shell2.mtx.graph	504855	8542010	ecology1.mtx.graph	1000000	1998000
af_shell3.mtx.graph	504855	8542010	ecology2.mtx.graph	999999	1997996
af_shell4.mtx.graph	504855	8542010	engine.mtx.graph	143571	2281251
af_shell5.mtx.graph	504855	8542010	F1.mtx.graph	343791	13246661
af_shell6.mtx.graph	504855	8542010	F2.mtx.graph	71505	2611390
af_shell7.mtx.graph	504855	8542010	Fault_639.mtx.graph	638802	13987881
af_shell8.mtx.graph	504855	8542010	filter3D.mtx.graph	106437	1300371
af_shell9.mtx.graph	504855	8542010	G3_circuit.mtx.graph	1585478	3037674
apache2.mtx.graph	715176	2051347	Ga10As10H30.mtx.graph	113081	3001276
BenElechi1.mtx.graph	245874	6452311	Ga19As19H42.mtx.graph	133123	4375858
bmw3_2.mtx.graph	227362	5530634	Ga3As3H12.mtx.graph	61349	2954799
bmw7st_1.mtx.graph	141347	3599160	Ga41As41H72.mtx.graph	268096	9110190
bmwcra_1.mtx.graph	148770	5247616	GaAsH6.mtx.graph	61349	1660230
boneS01.mtx.graph	127224	3293964	gas_sensor.mtx.graph	66917	818224
boyd1.mtx.graph	93279	558985	Ge87H76.mtx.graph	112985	3889605
c-73.mtx.graph	169422	554926	Ge99H100.mtx.graph	112985	4169205
c-73b.mtx.graph	169422	554926	gsm_106857.mtx.graph	589446	10584739
c-big.mtx.graph	345241	997885	H2O.mtx.graph	67024	1074856
cant.mtx.graph	62451	1972466	helm2d03.mtx.graph	392257	1174839
case39.mtx.graph	40216	516021	hood.mtx.graph	220542	5273947
case39_A_01.mtx.graph	40216	516021	IG5-17.mtx.graph	30162	1034600
case39_A_02.mtx.graph	40216	516026	invextr1_new.mtx.graph	30412	906915
case39_A_03.mtx.graph	40216	516026	kkt_power.mtx.graph	2063494	6482320
case39_A_04.mtx.graph	40216	516026	Lin.mtx.graph	256000	755200
case39_A_05.mtx.graph	40216	516026	mario002.mtx.graph	389874	933557
case39_A_06.mtx.graph	40216	516026	mixtank_new.mtx.graph	29957	982542
case39_A_07.mtx.graph	40216	516026	mouse_gene.mtx.graph	45101	14461095
case39_A_08.mtx.graph	40216	516026	msdoor.mtx.graph	415863	9912536
case39_A_09.mtx.graph	40216	516026	m_t1.mtx.graph	97578	4827996
case39_A_10.mtx.graph	40216	516026	nasasrb.mtx.graph	54870	1311227
case39_A_11.mtx.graph	40216	516026	nd12k.mtx.graph	36000	7092473
case39_A_12.mtx.graph	40216	516026	nd24k.mtx.graph	72000	14321817
case39_A_13.mtx.graph	40216	516026	nlpkkt80.mtx.graph	1062400	13821136
cf1.mtx.graph	70656	878854	offshore.mtx.graph	259789	1991442
cf2.mtx.graph	123440	1482229	oilpan.mtx.graph	73752	1761718
CO.mtx.graph	221119	3722469	parabolic_fem.mtx.graph	525825	1574400
consph.mtx.graph	83334	2963573	pdb1HYS.mtx.graph	36417	2154174
cop20k_A.mtx.graph	99843	1262244	pwtck.mtx.graph	217918	5708253
crankseg_1.mtx.graph	52804	5280703	qa8fk.mtx.graph	66127	797226
crankseg_2.mtx.graph	63838	7042510	qa8fm.mtx.graph	66127	797226
ct20stif.mtx.graph	52329	1323067			

file	n	m
s3dkq4m2.mtx.graph	90449	2365221
s3dkt3m2.mtx.graph	90449	1831506
shipsec1.mtx.graph	140874	3836265
shipsec5.mtx.graph	179860	4966618
shipsec8.mtx.graph	114919	3269240
ship_001.mtx.graph	34920	2304655
ship_003.mtx.graph	121728	3982153
Si34H36.mtx.graph	97569	2529405
Si41Ge41H72.mtx.graph	185639	7412813
Si87H76.mtx.graph	240369	5210631
SiO.mtx.graph	33401	642127
SiO2.mtx.graph	155331	5564086
sparsine.mtx.graph	50000	749494
StocF-1465.mtx.graph	1465137	9770126
t3dh.mtx.graph	79171	2136467
t3dh_a.mtx.graph	79171	2136467
thermal2.mtx.graph	1228045	3676134
thread.mtx.graph	29736	2220156
tmt_sym.mtx.graph	726713	2177124
TSOPF_FS_b162_c3.mtx.graph	30798	896688
TSOPF_FS_b162_c4.mtx.graph	40798	1193898
TSOPF_FS_b300.mtx.graph	29214	2196173
TSOPF_FS_b300_c1.mtx.graph	29214	2196173
TSOPF_FS_b300_c2.mtx.graph	56814	4376395
TSOPF_FS_b39_c19.mtx.graph	76216	979241
TSOPF_FS_b39_c30.mtx.graph	120216	1545521
turon_m.mtx.graph	189924	778531
vanbody.mtx.graph	47072	1144913
x104.mtx.graph	108384	5029620